Manual Técnico – Proyecto 2 “MinorC”

Requerimientos

Las herramientas y aplicaciones que fueron necesarias para el desarrollo del proyecto son las siguientes:

- Python 3.8.3 64- bit.
- Graphviz y el módulo para Python “graphviz-python 2.32.0 (la ruta del comando ‘dot’ debe de estar en las variables de entorno).
- QtDesigner y los módulos de PyQt5 (instalado por medio del comando pip install pyqt5).
- PLY 3.11 (instalado por medio del comando pip install ply).
- Visual Studio Code 1.46.0 como editor y debugger.
- Se utilizó el intérprete para Augus (proyecto 1) del compañero Haroldo Arias, el repositorio de su proyecto es el siguiente: https://github.com/harias25/Augus

Explicación del proyecto

El proyecto se enfoca en generar la traducción de un lenguaje de alto nivel, en este caso es MinorC (un lenguaje inspirado en C con instrucciones similares), a código de tres direcciones para que un intérprete de un lenguaje de bajo nivel, en este caso Augus (un lenguaje inspirado en PHP con instrucciones y funcionamiento similar), se encargue de realizar las operaciones y resolver la lógica de un programa. Por ello es necesario realizar un análisis sintáctico del lenguaje y a partir de él se genere la traducción de un código optimizado para su uso final.

Procedimientos:

Como fue mencionado previamente, se debe de realizar el análisis sintáctico del lenguaje de alto nivel. Fue utilizada la librería PLY por lo que es necesario definir varias expresiones regulares que funcionan como tokens para las palabras reservadas y caracteres del lenguaje. Estos se pueden encontrar en el archivo Tokens.py dentro del paquete MinorC.

```
def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'ID')
    return t

def t_DECIMAL(t):
    r'\d+\.\d+'
    return t

def t_INTEGER(t):
    r'\d+'
    return t

def t_CHARACTER(t):
    r'(\".\"|\'.\')'
    return t
```

Figura 1. Expresiones regulares de Tokens.

Una vez definidos los Tokens que serán utilizados en el análisis también se establecen otras clases para el manejo de las instrucciones que el lenguaje MinorC otorga. Estas pueden ser encontradas en el archivo Instructions.py dentro del paquete MinorC.

```
class IfElse:
    def __init__(self, expression, instruction, else_instruction):
        self.expression = expression
        self.instruction = instruction
        self.else_instruction = else_instruction


class Switch:
    def __init__(self, expression, case_list):
        self.expression = expression
        self.case_list = case_list


class Case:
    def __init__(self, expression, instructions):
        self.expression = expression
        self.instructions = instructions
```

Figura 2. Clases para las instrucciones de MinorC.

Estas clases son utilizadas para guardar los parámetros que conforman a cada instrucción; estos están bien definidos en el archivo.

También se aprovechó el análisis para generar el Árbol Abstracto de Sintaxis (AST) con la ayuda del paquete Graphviz. Este es creado en el archivo AscendentParser.py dentro del paquete MinorC. Al observar las reglas semánticas de las producciones de la gramática se generan índices para los nodos que representan la información de cada instrucción. Por lo que son utilizados para relacionar un nodo con una instrucción y así tener acceso a la instrucción por medio del nodo a medida que se va ascendiendo en el análisis.

La función ‘add_to_node’ y ‘get_from_node’ son las encargadas de hacer esto posible. Ya que cada nodo es almacenado en un diccionario, teniendo como clave única al índice de cada nodo y como valor al objeto instrucción.

```
def add_to_node(key, value):
    global ast_nodes
    ast_nodes[key] = value


def get_from_node(key):
    global ast_nodes
    return ast_nodes.get(key, None)
```

Figura 3. Métodos add_to_node y get_from_node.

Con estos métodos y clases ya es posible trabajar con las reglas semánticas de la gramática, teniendo como ejemplo la producción: BINARY -> EXPRESSION operador EXPRESSION.

```
def p_binary(p):
    '''binary           :   expression S_SUM expression
                        |   expression S_SUBS expression
                        |   expression S_ASTERISK expression
                        |   expression S_SLASH expression
                        |   expression S_PERCENTAGE expression
                        |   expression OP_AND expression
                        |   expression OP_OR expression
                        |   expression OP_COMPARISSON expression
                        |   expression OP_DISTINCT expression
                        |   expression OP_LESS_EQUAL expression
                        |   expression OP_GREATER_EQUAL expression
                        |   expression S_LESS expression
                        |   expression S_GREATER expression
                        |   expression S_AMPERSAND expression %prec OPB_AND
                        |   expression OPB_OR expression
                        |   expression OPB_XOR expression
                        |   expression OPB_L_SHIFT expression
                        |   expression OPB_R_SHIFT expression'''
    node_index = node_inc()
    dot.node(node_index, p.slice[2].value)
    dot.edge(node_index, p[1])
    dot.edge(node_index, p[3])
    operand1 = get_from_node(p[1])
    operand2 = get_from_node(p[3])
    new_binary = Binary(p.slice[2].value, operand1, operand2)
    add_to_node(node_index, new_binary)
    p[0] = node_index
```

Figura 4. Producción Binary y sus reglas semánticas para la gramática de MinorC.

Una vez finalizado el análisis de MinorC se procede a realizar la traducción. Para ello se utilizan los objetos tipo Instrucción que se generaron en el análisis previo. Con ello se obtiene una lista de cada instrucción con sus parámetros correspondientes y se estandarizan los métodos para cada instrucción. Estos algoritmos pueden ser encontrados en el archivo Translater.py dentro del paquete MinorC.